Beginning Git and GitHub

A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer

Mariot Tsitoara

Apress®

Beginning Git and GitHub

A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer

Mariot Tsitoara

Apress®

Beginning Git and GitHub

Mariot Tsitoara
Antananarivo, Madagascar

ISBN-13 (pbk): 978-1-4842-5312-0 ISBN-13 (electronic): 978-1-4842-5313-7
https://doi.org/10.1007/978-1-4842-5313-7

Managing Director, Apress LLC: Welmoed Spahr
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Coordinating Editor: Nancy Chen

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484253120. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to the generous people that made
the Git community such an awesome environment to work within.
You have helped create one of the most useful tools in the tech world.
Thank you!

Table of Contents

About the Author

Mariot Tsitoara is a Python and JavaScript developer with a passion for the Open Web and Data. He has been involved with Mozilla as a rep and a tech speaker since 2015 and has spoken extensively about Open Source and new technologies, including Rust, WebVR, and WebAssembly. Currently based in Bordeaux, he is constantly coding small, specialized tools for education. You can find him on Twitter @mariot_tsitoara.

About the Technical Reviewer

Alexander Chinedu Nnakwue has a background in Mechanical Engineering from the University of Ibadan, Nigeria, and has been a frontend developer for over 3 years working on both web and mobile technologies. He also has experience as a technical author, writer, and reviewer. He enjoys programming for the Web, and occasionally, you can also find him playing soccer. He was born in Benin City and is currently based in Lagos, Nigeria.

Acknowledgments

I'd like to thank my parents, Marie Jeanne and Tsitoara, for the amazing opportunities that they have given to me. Without their help and sacrifices, I would not be where I am today.

Thanks a lot also to my brothers and sisters, Alice, Elson, Thierry, Eliane, Annick, and Mamitiana, for being such amazing role models and for their constant support. To all my friends, Christino, Laza, Miandry, Mihaja, Miora, and Rindra, with whom I grew up and who taught me so much, I dedicate this book to you.

Almost everything I know about Git was taught to me by my coworkers. Thank you for being so helpful and a joy to work with.

This book wouldn't have seen the light of day if not for the amazing guidance of Nancy, Alexander, Louise, and Jim. Thank you so much!

Introduction

This book was written with a clear goal in mind: to be the book that I needed to read when I started my career in tech. Each chapter was crafted so that you will only be taught what you need to know as a beginner. It isn't a full reference book, but it can get you far enough to have a big impact on your career.

After reading this book, you will have the best tools for Version Control and Project Management.

Who is this book for

The targeted audience of this book is the absolute beginner with Git and GitHub and the people who have used them a little but want to know more. If you are searching for the best way to quick-start in the right direction, this book is for you.

How to use this book

Git is a very easy tool to learn, but you need to work with it to get the hang of it. The best way to learn is to directly use it on one of your real projects. Just reading the book and not doing any of the exercises will lengthen your learning curve.

PART I

Version Control with Git

CHAPTER 1

Version Control Systems

This is our first jump into Version Control Systems (VCSs). By the end of this chapter, you should know about Version Control, Git, and its history. The main objective is to know in which situations is Version Control needed and why Git is a safe choice.

What is Version Control?

As the name implies, Version Control is about the management of multiple versions of a project. To manage a version, each change (addition, edition, or removal) to the files in a project must be tracked. Version Control records each change made to a file (or a group of files) and offers a way to undo or roll back each change.

For an effective Version Control, you have to use tools called Version Control Systems. They help you navigate between changes and quickly let you go back to a previous version when something isn't right.

One of the most important advantages of using Version Control is teamwork. When more than one person is contributing to a project, tracking changes becomes a nightmare, and it greatly increases the probability of overwriting another person's changes. With Version Control, multiple people can work on their copy of the project (called branches) and only merge those changes to the main project when they (or the other team members) are satisfied with the work.

Note This book was written from a developer point of view, but everything in it applies to any text files, not just code. Version Control Systems can even track changes to many non-text files like images or Photoshop files.

M. Tsitoara, *Beginning Git and GitHub*, https://doi.org/10.1007/978-1-4842-5313-7_1

Why do you need one?

Have you ever worked on a text project or on a code that requires you to recall the specific changes made to each file? If yes, how did you manage and control each version? Maybe you tried to duplicate and rename the files with suffixes like “review,” “fixed,” or “final”? Figure 1-1 shows that kind of Version Control.

Figure 1-1. *Gimp files with suffixes like “final,” “final (copy),” and “reviewed”*

The figure shows what many people do to deal with file changes. As you can see, this has the potential to go out of hands very quickly. It is very easy to forget which file is which and what has changed between them.

To track versions, one idea is to compress the files and append timestamps to the names so that the versions are arranged by date of creation. Figure 1-2 shows that kind to version tracking.

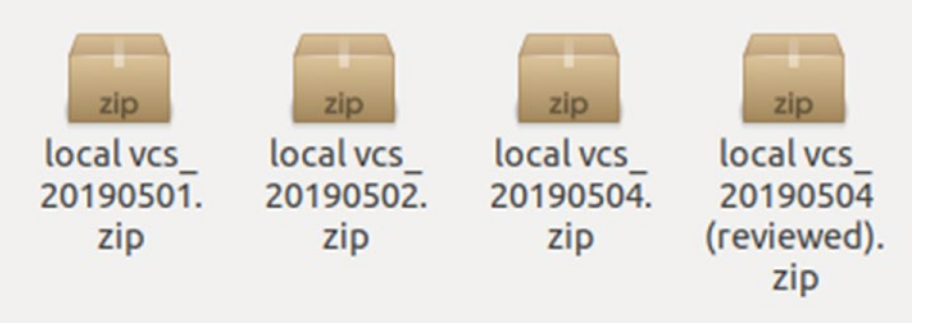

Figure 1-2. *Compressed version files sorted by dates*

The solution shown in Figure 1-2 appears to be the perfect system until you realize that even though the versions are tracked, there is no way to know what are the contents and descriptions of each version.

To remediate that situation, some developers use a solution like the one showed in Figure 1-3, which is to put the change summary of each version in a separate file.

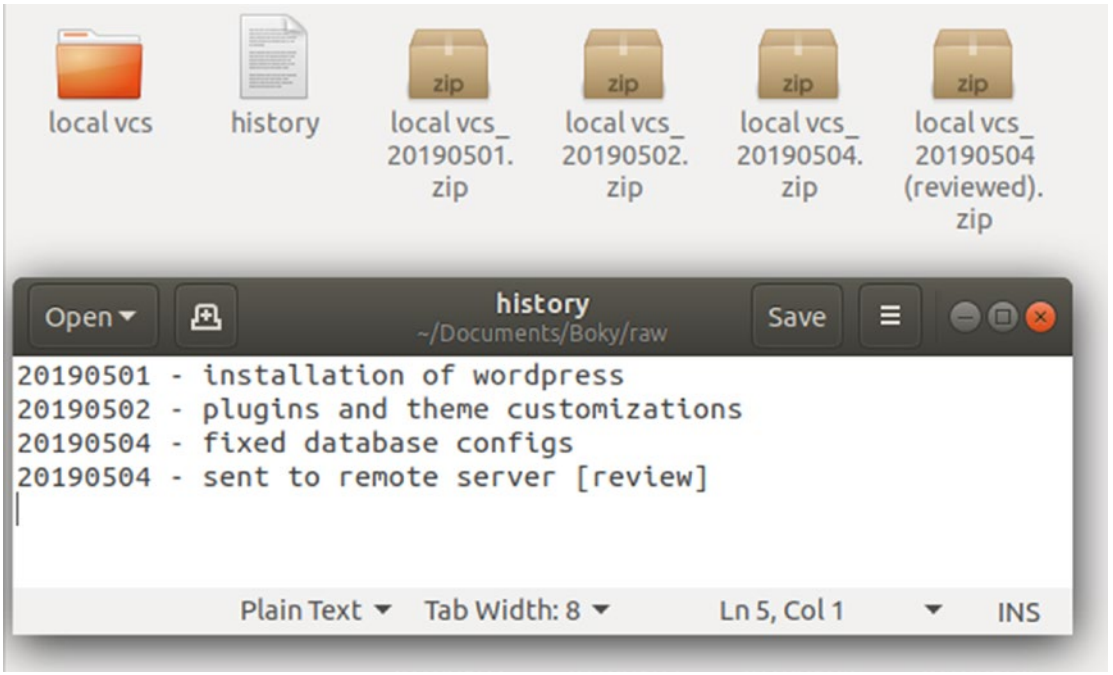

Figure 1-3. *A separate file where each version is tracked*

As Figure 1-3 shows, a separate file accompanies the project folder with a short description of the change made. Also note the many compressed files which contain the previous versions of the project.

That should do it, right? Not quite, you would still need a way to compare each version and every file change. There is no way to do this in that system; you just need to memorize everything you did. And if the project gets big, the folder just gets bigger with each version.

What happens when another developer or writer joins your team? Would you email each other the files or versions you edited? Or work on the same remote folder? In the last case, how would you know who is working on which file and what changed?

And lastly, have you ever felt the need to undo a change you made years ago without breaking everything in the process? An unlimited and all-powerful ctrl-z?

All those problems are solved by using a Version Control System or VCS. A VCS tracks each change you made to every file of your project and provides a simple way to

compare and roll back those changes. Each version of the project is also accompanied by the description of the changes made along with a list of the new or edited files. When more people join the project, a VCS can show exactly who edited a particular file on a specific time. All of that makes you gain precious time for your project because you can focus on writing instead of spending time tracking each change. Figure 1-4 shows a versioned project managed by Git.

As shown in Figure 1-4, a versioned project combines all the solutions we tried in this chapter. There are the change descriptions, the teamwork, and the edit dates.

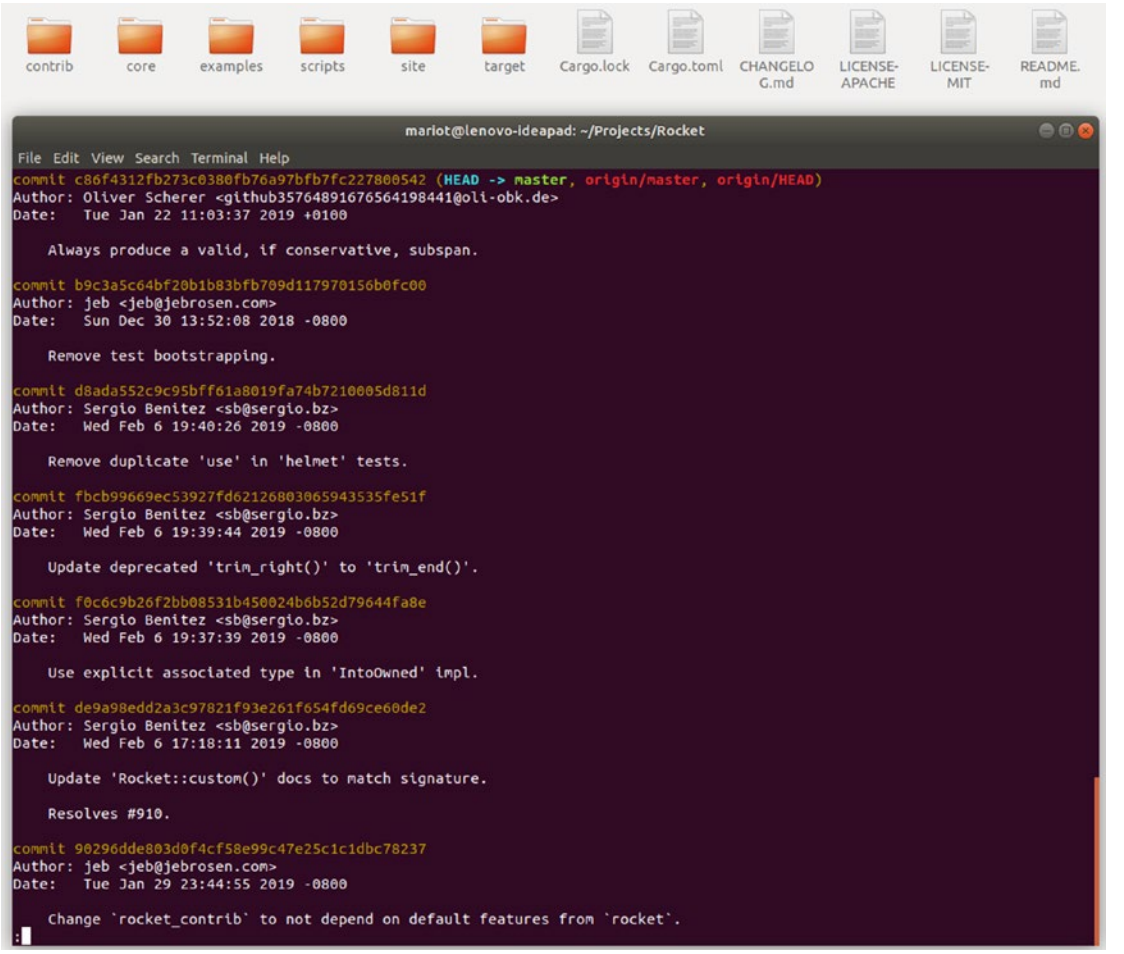

Figure 1-4. *A project versioned by Git*

Let's find out more about Version Control Systems.

What are the choices?

There are many flavors of Version Control Systems, each with their own advantages and shortcomings. A VCS can be local, centralized, or distributed.

Local Version Control Systems

These are the first VCSs created to manage source code. They worked by tracking the changes made to files in a single database that was stored locally. This means that all the changes were kept in a single computer and if there were problems, all the work were lost. This also means that working with a team was out of the question.

One of the most popular local VCSs was Source Code Control System or SCCS, which was free but closed source. Developed by AT&T, it was wildly used in the 1970s until Revision Control System or RCS was released. RCS became more popular than SCCS because it was Open Source, cross-platform, and much more effective. Released in 1982, RCS is currently maintained by the GNU Project. One of the drawbacks of these two local VCSs was that they only worked on a file at a time; there was no way to track an entire project with them.

To help you visualize how it works, here's Figure 1-5 which shows an illustration of a simple local VCS.

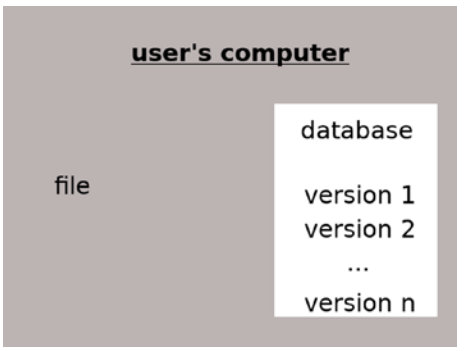

Figure 1-5. *How a local VCS works*

As you can see in Figure 1-5, everything is on the user's computer, and only one file is tracked. The versioning is stored in a database managed by the local VCS.

Centralized Version Control Systems

Centralized VCS (CVCS) works by storing the change history on a single server that the clients (authors) can connect to. This offers a way to work with a team and also a way to monitor the general pace of a project. They are still popular because the concept is so simple and it's very easy to set up.

The main problem was that, like local VCS, a server error can cost the team all their work. A network connection was also required since the main project was stored in a remote server.

You can see in Figure 1-6 how it works.

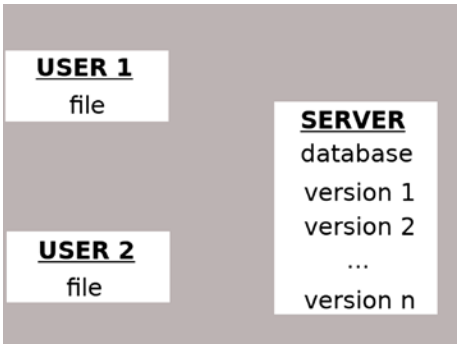

Figure 1-6. *How a centralized VCS works*

Figure 1-6 shows that a centralized VCS works similarly to a local VCS, but the database is stored in a remote server.

The main problem faced by team using a centralized VCS is that once a file is being used by someone, that file is locked and the other team members can't work on it. Thus, they had to coordinate between themselves just to modify a single file. This creates a lot of delays in development and is generally source to a lot of frustration for contributors. And the more members are on the team, the more problems arise.

In an effort to counter the problems of local VCS, Concurrent Version System or CVS was developed. It was Open Source and could track multiple sets of files instead of a single file. Many users could also work on the same file at the same time, hence the "concurrent" in the name. All the history was stored in a remote repository, and the users would keep up with the changes by checking out the server, meaning copying the contents of the remote database to their local computers.

Apache Subversion or SVN was developed in 2000 and could be everything that CVS could, with a bonus: it could track non-text files. One of the main advantages of SVN was that instead of tracking a group of files like the previous VCS, it tracks the entire project. So, it is essentially tracking the directory instead of files. That means that the renaming, adding and removing are also tracked. This made SVN, along with it being Open Source, a very popular VCS; and it is still wildly used today.

Distributed Version Control Systems

Distributed VCS works nearly the same as centralized VCS but with a big difference: there is no main server that holds all the history. Each client has a copy of the repository (along with the change history) instead of checking out a single server.

This greatly lowers the chance of losing everything as each client has a clone of the project. With a distributed VCS, the concept of having a "main server" gets blurred because each client essentially has all the power within their own repository. This greatly encouraged the concept of "forking" within the Open Source community. Forking is the act of cloning a repository to make your own changes and have a different take on the project. The main benefit of forking is that you could also pull changes from other repositories if you see fit (and others can do the same with your changes).

A distributed Version Control System is generally faster than the other types of VCS because it doesn't need a network access to a remote server. Nearly everything is done locally. There is also a slight difference with how it works: instead of tracking the changes between versions, it tracks all changes as "patches." This means that those patches can be freely exchanged between repositories, so there is no "main" repository to keep up with.

Figure 1-7 shows how a distributed VCS works.

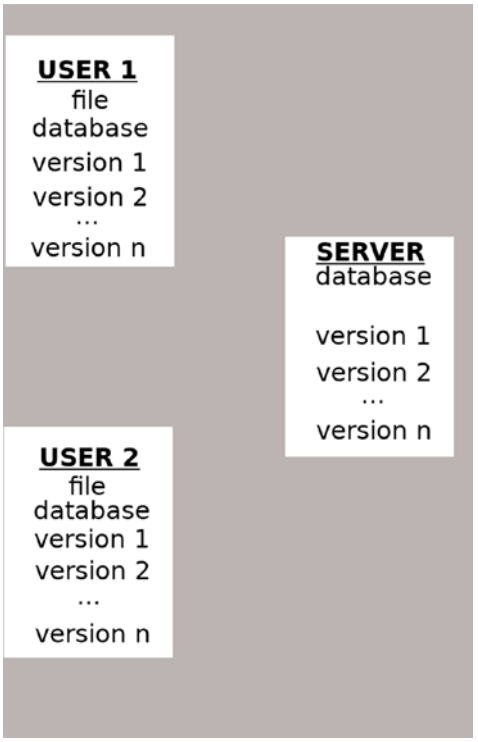

Figure 1-7. *How a distributed VCS works*

Note By looking at Figure 1-7, it is tempting to conclude that there is a main server that the users are keeping up with. But it isn't the case with a distributed VCS, it is only a convention that many developers follow to have a better workflow.

BitKeeper SCM was a proprietary distributed VCS released in 2000 which, like SCCS in the 1970s, was closed source. It had a free "Community Version" that lacked many of the big features of BitKeeper SCM, but since it was one of the first distributed VCSs, it was pretty popular even in the Open Source community. This popularity of BitKeeper plays a big role in the creation of Git. It is now an Open Source software, after having its source code released under the Apache License in 2016. You can find the current BitKeeper project on `www.bitkeeper.org/`; the development has slowed down, but there is still a community contributing to it.

What is Git?

Remember the proprietary distributed Version Control System BitKeeper SCM from the last section? Well, the Linux kernel developers used it for their development. The decision to use it was wildly regarded as a bad move and made many people unhappy. Their fears were confirmed in 2005 when BitKeeper SCM stopped being free. Since it was closed source, the developers lost their favorite Version Control System. The community (led by Linus Torvalds) had to find another VCS, and since an alternative was not available, they decided to create their own. Thus, Git was born.

Since Git was made to replace BitKeeper SCM, it worked generally the same with a few tweaks. Like BitKeeper SCM, Git is a distributed Version Control System, but it is faster and works better with large projects. The Git community is very active, and there are many contributors involved in its development; you can find more about Git on https://git-scm.com/. The features of Git and how it works are explained later in this section.

What can Git do?

Remember all those problems we tried to solve at the beginning of this chapter? Well, Git can solve them all. It can even solve problems you didn't know you had!

First, it works great with tracking changes. You can

- Go back and forth between versions
- Review the differences between those versions
- Check the change history of a file
- Tag a specific version for quick referencing

Git is also a great tool for teamwork. You can

- Exchange "changesets" between repositories
- Review the changes made by others

One of the main features of Git is its Branching system. A branch is a copy of a project which you can work on without messing with the repository. This concept has been around for some time, but with Git, it is way faster and more efficient. Branching also comes along with Merging, which is the act of copying the changesets done in a branch

back to the source. Generally, you create a branch to create or test a new feature and merge that branch back when you are satisfied with the work.

There is also a simple concept that you might use a lot: Stashing. Stashing is the act of safely putting away your current edits so that you have clean environment to work on something completely different. You might want to use stashing when you are playing around or testing a feature but need to work on a new feature in priority. So, you stash your changes away and begin to write that feature. After you are done, you can get your changes back and apply them to your current working environment.

As a little appetizer, here are some of the Git commands you will learn in this book:

```
$ git init     # Initialize a new git database
$ git clone    # Copy an existing database
$ git status   # Check the status of the local project
$ git diff     # Review the changes done to the project
$ git add      # Tell Git to track a changed file
$ git commit   # Save the current state of the project to database
$ git push     # Copy the local database to a remote server
$ git pull     # Copy a remote database to a local machine
$ git log      # Check the history of the project
$ git branch   # List, create or delete branches
$ git merge    # Merge the history of two branches together
$ git stash    # Keep the current changes stashed away to be used later
```

As you can see, the commands are pretty self-explanatory. Don't worry about knowing all of them by heart; you will retain them one by one when we will properly begin the learning. And you will not also use them all the time, you will mostly use git add and git commit. You will learn about each command, but we will focus on the commands that you will likely use in a professional setting. But before that, let's see the inner working of Git.

How does Git work?

Unlike many Version Control Systems, Git works with Snapshots, not Differences. This means that it does not track the difference between two versions of a file, but takes a picture of the current state of the project.

This is why Git is very fast compared to other distributed VCSs; it is also why switching between versions and branches is so fast and easy.

Remember how a centralized Version Control System works? Well, Git is the complete opposite. You don't need to communicate with a central server get work done. Since Git is a distributed VCS, every user has their own fully fledged repository with their own history and changesets. Thus, everything is done locally except the sharing of patches or changesets. Like previously said, a central server is not needed; but many developers use one as convention as it is easier to work that way.

Speaking of patch sharing, how does Git know which changesets are whose? When Git takes a snapshot, it performs a checksum on it; so, it knows which files were changed by comparing the checksums. This is why Git can track changes between files and directories easily, and it also checks for any file corruption.

The main feature of Git is its "Three States" system. The states are the working directory, the staging area, and the git directory:

- The working directory is just the current snapshot that you are working on.
- The staging area is where modified files are marked in their current version, ready to be stored in the database.
- The git directory is the database where the history is stored.

So, basically Git works as follows: you modify the files, add each file you want to include in the snapshot to the staging area (git add), then take the snapshot and add them to the database (git commit). For the terminology, we call a modified file added to the staging area "staged" and a file added to the database "committed." So, a file goes from "modified" to "staged" to "committed."

What is the typical Git workflow?

To help you visualize all that we talked about in this section, here is a little demo of what a typical workflow using Git is like. Don't worry if you don't understand everything right now; the next chapters will get you set up.

This is your first day of work. You are tasked to add your name to an existing project description file. Since this is your first day, a senior developer is there to review your code.

The first thing you should do is get the project's source code. Ask your manager for the server where the code is stored. For this demo, the server is GitHub, meaning that the

Git database is stored on a remote server hosted by GitHub and you can access it by URL or directly on the GitHub web site. Here, we are going to use the clone command to get the database, but you could also just download the project from the GitHub web site. You will get a zip file containing and the project files with all its history.

So, you clone the repository to get the source code by using the "clone" command.

```
git clone https://github.com/mariot/thebestwebsite.git
```

Git then downloads a copy of the repository in the current directory you are working from. After that, you can enter the new directory and check its contents as shown in Figure 1-8.

Figure 1-8. *The contents of the repository is shown*

If you want to check the recent changes made to the project, you can use the "log" command to show the history. Figure 1-9 shows an example of that.

```
Mariot@lenovo-ideapad MINGW64 ~/Documents/Boky/raw/thebestwe
$ git log
commit 0cc01f912449ed913c9f48673a4b450a66951f31 (HEAD -> mas
Author: Denys Vitali <denys@denv.it>
Date:   Fri Jan 18 17:44:45 2019 +0100

    Add Hugo Theme references

    Reference: https://github.com/hugomodo/hugomodo-best-mot

commit 033eb62a526e4ffd9c73257ab37e76c9d484cd74
Author: Denys Vitali <denys@denv.it>
Date:   Thu Jan 10 10:46:28 2019 +0100

    Fix #31, add inverted-contrast mode

commit 74452d4c8cacb2dcad4431532eb99ccac4b00eac
Merge: 13e4f7e 6c3ba31
Author: Denys Vitali <denys@denv.it>
Date:   Mon Nov 12 10:12:39 2018 +0100

    Merge pull request #30 from numbermaniac/patch-1

    create -> created

commit 6c3ba31b95190fdaecf95b9af2b9d2f5554d7203
Author: numbermaniac <numbermaniac@users.noreply.github.com>
Date:   Sun Nov 11 11:22:45 2018 +1100

    create -> created
```

Figure 1-9. *A typical Git history log*

Nice! Now you should create a new branch to work on so that you don't mess up with the project. You can create a new branch by using the "branch" command and checking it out with the "checkout" command.

```
git branch add-new-dev-name-to-readme
git checkout add-new-dev-name-to-readme
```

Now that the new branch is created, you can begin to modify the files. You can use whatever editor you want; Git will track all the changes via checksums. Now that you made the necessary changes, it is time to put them on the staging area. As a reminder, the staging area is where you put modified codes that are ready to be snapshotted. If we modified the "README.md" file, we can add it to the staging area by using the "add" command.

```
git add README.md
```

You don't have to add every file you modified to the staging area, only those which you want to be accounted in the snapshot. Now that the file is staged, it is time to "commit" it or putting its change in the database. We do this by using the command "commit" and attaching a little description with it.

```
git commit -m "Add Mariot to the list of developers"
```

And that's it! The changes you made are now in the database and safely stored. But only on your computer! The others can't see your work because you worked on your own repository and on a different branch. To show your work to others, you have to push your commits to the remote server. But you have to show the code to the senior dev first before making a push. If they are okay with it, you can merge your branch with the main snapshot of the project (called the master branch). So first you must navigate back to the master branch by using the "checkout" command.

```
git checkout master
```

You are now on the master branch, where all the team's work is stored. But the time you worked on your fix, the project may have changed, meaning that a team member may have changed some files. You should retrieve those changes before committing your own changes to master. This will limit the risk of "conflicts" which can happen when two or more contributors change the same file. To get the changes, you have to pull the project from the remote server (also called origin).

```
git pull origin master
```

Even if another coworker changed the same file as you, the risk of conflicts is low because you only modified a line. Conflicts only arise when the same line has been modified by multiple people. If you and your coworkers changed different parts of the file, everything is okay.

Now that we kept up with the current state of the project, it's time to commit our version to master. You can merge your branch with the "merge" command.

```
git merge add-new-dev-name-to-readme
```

Now that the commit has been merged back to master, it is time to push the changes to the main server. We do that by using to "push" command.

```
git push
```